

bris: un sistema distribuito per giocare a briscola

Progetto del modulo di laboratorio dei corsi di SO A/B 2013/14

Indice

1	Introduzione	1
1.1	Materiale in linea	2
1.2	Struttura del progetto e tempi di consegna	2
1.3	Valutazione del progetto	2
2	Il progetto: bris	3
3	Il server	4
4	Il client	5
5	Protocollo di interazione	6
5.1	Formato dei messaggi	6
5.2	Messaggi da Client a Server	7
5.3	Messaggi da Server a Client	7
6	Lo script bristat	8
7	Makefile	8
8	Istruzioni	9
8.1	Materiale fornito dai docenti	9
8.2	Cosa devono fare gli studenti	9
9	Parti Opzionali	9
10	Codice e documentazione	9
10.1	Vincoli sul codice	9
10.2	Formato del codice	10
10.3	Relazione	10

1 Introduzione

Il modulo di Laboratorio di Programmazione di Sistema del corso di Sistemi Operativi e Laboratorio (277AA) prevede lo svolgimento di un progetto individuale suddiviso in tre frammenti. Questo documento descrive la struttura

complessiva del progetto e dei vari frammenti che lo compongono utilizzando gli strumenti, le tecniche e le convenzioni presentati durante il corso.

1.1 Materiale in linea

Tutto il materiale relativo al corso può essere reperito sul sito Web:

<http://www.cli.di.unipi.it/doku/doku.php/informatica/sol/laboratorio13/>

Il sito verrà progressivamente aggiornato con le informazioni riguardanti il progetto (es. FAQ, suggerimenti, avvisi), il ricevimento e simili. Il sito è un Wiki e gli studenti possono registrarsi per ricevere automaticamente gli aggiornamenti delle pagine che interessano maggiormente. In particolare consigliamo a tutti di registrarsi alla pagina delle 'FAQ' e degli 'Avvisi Urgenti'.

Eventuali chiarimenti possono essere richiesti consultando i docenti del corso durante l'orario di ricevimento, le ore in laboratorio e/o per posta elettronica.

1.2 Struttura del progetto e tempi di consegna

Il progetto deve essere sviluppato dallo studente individualmente e può essere consegnato entro il **15 Gennaio 2014**.

La consegna del progetto avviene *esclusivamente* inviando per posta elettronica il tar creato dal target `consegna3` del `Makefile` contenuto nel kit di sviluppo del progetto. Il tar deve essere allegato ad un messaggio con soggetto

`lso13: consegna progetto finale`

ed inviato a `susanna.pelagatti@gmail.com`. Le consegne sono seguite da un messaggio di conferma da parte del docente all'indirizzo di mail da cui la consegna è stata effettuata. Se la ricezione non viene confermata entro 3/4 giorni lavorativi, contattare il docente per e-mail.

I progetti che non rispettano il formato o non generati con il target `consegna3` non verranno accettati. Si prega di controllare che tutti i file necessari alla corretta compilazione ed esecuzione del progetto siano presenti nel tar prima di inviarlo.

La data ultima di consegna è il 15/01/2014. Dopo questa data gli studenti dovranno svolgere il nuovo progetto previsto per il corso 2013/14.

Il progetto è suddiviso in tre frammenti. Gli studenti che consegnano una versione sufficiente di ogni frammento entro la data di scadenza specificata sul WEB accumulano dei bonus di 2 punti che contribuiscono al voto finale con le modalità illustrate nelle slide introduttive del corso (vedi sito).

1.3 Valutazione del progetto

Al progetto viene assegnata una valutazione da 0 a 30 di cui 6 punti esclusivamente determinati dalla *qualità della documentazione allegata*. La valutazione del progetto è effettuata in base ai seguenti criteri:

- motivazioni, originalità ed economicità delle scelte progettuali
- strutturazione del codice (suddivisione in moduli, uso di makefile e librerie etc.)
- efficienza e robustezza del software
- modalità di testing
- aderenza alle specifiche
- qualità del codice C e dei commenti
- chiarezza ed adeguatezza della relazione (vedi Sez. 10.3)

La prova orale tenderà a stabilire se lo studente è realmente l'autore del progetto consegnato e verterà su tutto il programma del corso e su tutto quanto usato nel progetto anche se non fa parte del programma del corso. Il voto dell'orale (da 0 a 30L) fa media con la valutazione del progetto per delineare il voto finale. In particolare, l'orale comprenderà

- una discussione delle scelte implementative
- l'impostazione e la scrittura di script bash e makefile
- l'impostazione e la scrittura di programmi C + PosiX non banali (sia sequenziali che concorrenti)
- domande su tutto il programma presentato durante il corso.

Casi particolari Gli studenti lavoratori iscritti alla laurea triennale in Informatica possono consegnare i due frammenti e il progetto finale in un'unica soluzione in qualsiasi momento dell'anno ed essere valutati con votazione da 0 a 30. In questo caso è necessaria la certificazione da consegnare al docente come da regolamento di ateneo.

Gli studenti che svolgono il progetto per le lauree di secondo livello sono invitati a contattare il docente.

Gli studenti iscritti ai vecchi ordinamenti (nei quali il voto di Lab. 4 contribuiva al voto di SO) avranno assegnato un voto in trentesimi che verrà combinato con il voto del corso di SO corrispondente secondo modalità da richiedere ai due docenti coinvolti.

2 Il progetto: bris

Lo scopo del progetto è lo sviluppo di un sistema distribuito per permettere a più utenti connessi di giocare a briscola¹. I valori delle carte sono:

- asso (A) 11 punti
- 3 (3) 10 punti
- Re (K) 4 punti
- Regina (Q) 3 punti
- Fante (J) 2 punti
- 2,4,5,6,7 (2,4,5,6,7) 0 punti

Il sistema è costituito da più istanze dei seguenti processi:

- **brsserver**: il processo server, che si occupa di registrare e cancellare gli utenti e di gestire le partite.
- **brsclient**: il processo client con cui è possibile registrarsi, cancellarsi e giocare una partita.

brsclient e **brsserver** sono i due processi che devono essere realizzati nel progetto didattico. I processi comunicano via socket AF_UNIX. Per rendere più semplice lo sviluppo ed il testing dell'applicazione sulle macchine del centro di calcolo per la didattica tutte le socket vengono create nella directory locale `./tmp` invece che in `/tmp` come sarebbe logico aspettarsi. Quest'ultima soluzione renderebbe infatti possibili interazioni indesiderate e fastidiose fra progetti sviluppati da utenti diversi sulla stessa macchina.

Inoltre, deve essere realizzato uno script bash (**bristat**) che analizza i file di log relativi alle partite come specificato in Sez. 6.

¹<http://it.wikipedia.org/wiki/Briscola>

3 Il server

`brssserver` viene attivato da shell con il comando

```
$ brssserver file_utenti -t
$ brssserver file_utenti
```

dove `file_utenti` è il file di testo che contiene gli utenti ammessi ad utilizzare il server e la password con cui devono collegarsi al servizio secondo il formato:

```
nomeutente:passwd
```

le informazioni dei diversi utenti sono separate da un `\n` (ritorno carrello). L'opzione `-t` indica l'attivazione del server in modalità testing, questa opzione serve a indicare che stiamo testando il server con test automatici e quindi i mazzi di carte devono essere generati in modo predicibile. La funzione di generazione del mazzo `thread safe`

```
mazzo_t* newMazzo_r (bool_t test);
```

viene fornita dai docenti nel kit del terzo frammento e deve essere invocata con parametro `TRUE` in modalità test e `FALSE` in normali esecuzioni.

Il server per ogni partita registra anche un file di log (`./BRS-npartita.log`) in cui per la partita numero `npartita` vengono registrati gli utenti coinvolti, le varie mani, e l'esito finale con il formato seguente:

```
nomeutente1:nomeutente2
BRISCOLA:Q
nomeutente1:2F#nomeutente2:5Q
nomeutente2:JP#nomeutente1:4C
...
WINS:nomeutente2
POINTS:74
```

Dove la prima riga specifica i due utenti che sono coinvolti nella partita, la seconda specifica il seme di briscola (`Q,C,P,F` quadri, cuori, picche, fiori) le linee successive denotano tutte le mani (si ricorda che chi vince una mano inizia la successiva). Nell'esempio `A` denota l'asso, `K,Q,J` denotano rispettivamente re, regina e fante e le altre carte sono denotate con caratteri da 2 a 7. Infine nelle ultime due linee del file abbiamo il nome dell'utente vincitore ed i punti totalizzati.

Il funzionamento del server è il seguente. Se il file passato come argomento esiste viene aperto in lettura e ne viene verificato il formato. Se il formato è corretto gli utenti abilitati vengono memorizzati utilizzando un albero binario di ricerca. L'albero e le funzioni per la sua manipolazione sono fissate e realizzate nel primo frammento del progetto. A questo punto si crea una socket `AF_UNIX`

```
./tmp/briscola.skt
```

se la socket esiste già all'avvio, il server fallisce. Su questa socket i client aprono le connessioni per interagire con il server.

Il server è multithreaded e tutte le operazioni richieste dagli utenti (registrazioni, cancellazioni, partite) devono essere eseguite dai thread worker in parallelo fra di loro. Inoltre deve essere presente un thread diverso dai worker che accetta le nuove richieste di connessione al servizio in parallelo con i worker.

Nel caso sia richiesto di effettuare una partita, in fase di autenticazione il client riceve una lista di possibili avversari. La lista comprende tutti gli utenti attualmente connessi che stanno aspettando una sfida. A questo punto può decidere se sfidare un utente della lista o mettersi a sua volta in attesa di essere sfidato. Se decide di attendere

una sfida, il worker segnala l'attesa sull'albero e termina. Se sfida un avversario la partita inizia immediatamente sotto la supervisione del worker che mescola le carte, le fornisce ai giocatori, comunica l'esito di ciascuna mano e le nuove carte pescate dal mazzo. Gioca sempre per primo in una partita il giocatore che ha lanciato la sfida. È il thread worker a gestire la scrittura del file di log corrispondente alla partita.

Quando il server è attivo nuovi utenti possono registrarsi, connettersi, oppure richiedere di essere cancellati. In qualsiasi momento, si può richiedere la scrittura su file della versione corrente albero degli utenti inviando un segnale **SIGUSR1**. Alla ricezione di tale segnale il server salva il contenuto dell'albero nel file **./brs.checkpoint** sovrascrivendolo se già presente. Il formato è quello definito per **file_utenti**.

Il server termina quando riceve un segnale di **SIGINT** o **SIGTERM**, in questo caso si attende la terminazione dei thread worker, in modo che vengano terminate le partite in corso, venga completata la scrittura sui file di log, venga aggiornato il file degli utenti registrati **file_utenti** e venga cancellata la socket del server.

Il protocollo di interazione fra client e server è descritto nella Sezione 5.

4 Il client

Il client **brsclient** è un comando Unix che viene invocato con formato

```
$ ./brsclient nomeuser pwd -r
$ ./brsclient nomeuser pwd -c
$ ./brsclient nomeuser pwd
```

dove **nomeuser** rappresenta l'utente che desidera connettersi e **pwd** è la password. Le opzioni **-r** e **-c** servono a registrare un nuovo utente e a cancellare un utente esistente rispettivamente. In questo caso il client richiede l'operazione al server, attende la risposta dell'esito, stampa l'esito e termina.

Se invece il client viene attivato senza opzioni, l'utente desidera ingaggiare una partita. Ad esempio supponiamo di invocare il client con

```
brsclient pippo ppllmm
```

In questo caso, il client come prima cosa tenta la connessione con il server. Se la connessione ha successo invia user e password per l'autenticazione. Se l'autenticazione ha successo il server risponde positivamente ed invia la lista dei possibili avversari (gli utenti connessi in attesa di una sfida). Il client deve proporre la scelta all'utente come in

```
Possibili avversari: minni:nonnapapera:pluto
WAIT per attendere una sfida
Scelta --?
```

se l'utente digita **WAIT** significa che preferisce attendere di essere sfidato, altrimenti deve digitare il nome di un avversario, in questo caso **minni**, **nonnapapera** o **pluto**. La scelta viene inviata al server ed in entrambi i casi il client si mette in attesa di un successivo messaggio di inizio partita, secondo il protocollo specificato nella Sez. 5. Quando il messaggio arriva il client visualizza il nome dell'avversario, il seme di briscola e le carte della prima mano

```
Giochiamo con minni
Briscola: F
Carte: 2Q 2F QP
Turno: pippo --?
```

dopodiché viene visualizzato il giocatore di turno (il primo turno è sempre per chi lancia la sfida, in questo caso **pip**po). Nel caso il turno sia dell'altro giocatore la carta giocata verrà notificata dal server mentre quando è il turno del giocatore **pip**po il client si metterà in attesa della carta giocata sullo standard input per inviarla al server. Ad esempio la partita precedente può continuare così

```
...
Turno: pippo --? 3Q
Turno: minni --> 2F
Carte: 2Q QF QP
Turno: minni -->
```

dove il punto interrogativo sottolinea l'attesa di un input da tastiera da parte dell'utente mentre la freccia indica che stiamo attendendo dal server la giocata dell'altro. Dopo la mano vengono stampate le tre carte in mano all'utente e il turno.

Alla fine della partita il client riceve dal server l'indicazione del vincitore e dei punti totalizzati. A questo punto informa l'utente con una stampa, come ad esempio

```
Vince minni con 74 punti!
Bye
```

e termina.

5 Protocollo di interazione

I server ed i client interagiscono con un socket *AF_UNIX*.

I client si connettono al server attraverso la socket pubblica `./tmp/briscola.sck`, creata all'avvio del server. I tipi descritti in questa sezione e i prototipi delle funzioni di comunicazione da realizzare per il secondo frammento di progetto sono contenuti nel file `comsock.h`. Va sottolineato che è richiesto che l'invio dei messaggi venga effettuato serializzando i campi della struttura su un array di caratteri ed effettuando una singola scrittura sulla socket. Questo è necessario per garantire l'atomicità della scrittura delle varie parti del messaggio in presenza di più invii contemporanei sulla stessa socket.

5.1 Formato dei messaggi

I messaggi scambiati fra server e client hanno la seguente struttura:

```
typedef struct {
    char type;
    unsigned int length;
    char * buffer;
} message_t;
```

Il campo `type` è un `char` (8 bit) che contiene il tipo del messaggio spedito. `type` può assumere i seguenti valori:

```
/** richiesta di registrazione utente */
#define MSG_REG          'R'
/** richiesta di cancellazione utente */
#define MSG_CANC         'Q'
/** richiesta di connessione al servizio */
#define MSG_CONNECT      'C'
/** attesa di un avversario */
#define MSG_WAIT         'W'
```

```

/** accettazione */
#define MSG_OK          'K'
/** rifiuto */
#define MSG_NO          'N'
/** errore */
#define MSG_ERR         'E'
/** comunicazione inizio gioco avversario, briscola, prima mano, turno*/
#define MSG_STARTGAME   'S'
/** comunicazione fine gioco */
#define MSG_ENDGAME     'Z'
/** comunicazione carta giocata */
#define MSG_PLAY        'P'
/** comunicazione nuova carta */
#define MSG_CARD        'A'

```

Il campo `length` è un `unsigned int` che indica il numero dei dati significativi all'interno del campo `buffer`. Il campo `length` vale 0 nel caso in cui il campo `buffer` non sia significativo.

5.2 Messaggi da Client a Server

Nei messaggi spediti dal client al server, il campo `type` può assumere i seguenti valori:

MSG_REG MSG_CANC messaggio per richiedere la registrazione o cancellazione di un utente.

Nel campo `buffer` è contenuto il nome dell'utente e la password, ad esempio `minni:ppwddd`. Il server risponderà con un `MSG_OK` se l'operazione è stata effettuata, `MSG_NO` se non è consentita, ad esempio la password non è corretta o l'utente non è presente (in questo caso il campo `buffer` contiene la motivazione da fornire all'utente), oppure `MSG_ERR` se si è verificato un errore (anche in questo caso il campo `buffer` può fornire più informazioni).

MSG_CONNECT Messaggio per richiedere la connessione al servizio e l'inizio partita, anche in questo caso si inviano utente e password nel campo `buffer`. Il server risponderà con un `MSG_OK` o `MSG_WAIT` se è possibile e con `MSG_NO` o `MSG_ERR` se non è possibile (come nel caso precedente). Nel caso il messaggio sia di tipo `MSG_OK` il server invia nel `buffer` anche la lista dei possibili avversari, come ad esempio `pippo:pluto:paperone`. Nel caso il messaggio sia di tipo `MSG_WAIT` non ci sono avversari disponibili e il client si deve mettere in attesa.

MSG_WAIT Messaggio per indicare che l'utente intende comunque aspettare uno sfidante.

MSG_OK Messaggio per indicare che l'utente vuole iniziare la partita, in questo caso il campo `buffer` contiene il nome dell'utente da sfidare, ad esempio `pippo`.

MSG_PLAY Messaggio per indicare la carta che si intende giocare, il `buffer` contiene due caratteri il primo per il valore ed il secondo per il seme, secondo il formato stringa specificato nel primo frammento, ad esempio `AC`.

5.3 Messaggi da Server a Client

Nei messaggi spediti dal server a client, il campo `type` può assumere i seguenti valori:

MSG_NO Messaggio di risposta negativa. Con questi messaggio il server segnala che non è permesso effettuare l'operazione, il campo `buffer` contiene la motivazione.

MSG_ERR Messaggio di errore server. Con questi messaggio il server segnala che l'operazione è permessa ma si è verificato un errore, il campo `buffer` contiene la descrizione dell'errore.

MSG_OK Messaggio di risposta positiva. Con questi messaggio il server segnala che l'operazione è stata effettuata con successo.

MSG_WAIT Non ci sono avversari disponibili per la partita.

MSG_STARTGAME Messaggio di inizio partita, si indicano l'avversario, la briscola, la prima mano di carte secondo un formato a scelta dello studente, ad esempio il buffer può contenere **Q:AFQQ3P:pluto**. Il primo turno di una partita è sempre per l'avversario che ha lanciato la sfida.

MSG_ENDGAME Messaggio di fine partita, si indicano l'utente che ha vinto ed i punti complessivi totalizzati dal vincitore, ad esempio **minni:74**.

MSG_PLAY Carta giocata, il buffer contiene la carta, ad esempio **QF**.

MSG_CARD Messaggio di inizio prossima mano. Con questo messaggio si indica l'utente che deve giocare la prossima mano e la nuova carta "pescata" dal mazzo. Ad esempio il buffer può contenere **t:AQ** oppure **a:AQ** dove **t** indica turno del giocatore che riceve il messaggio ed **a** indica il turno dell'altro giocatore.

6 Lo script bristat

I file di log possono essere esaminati utilizzando lo script **bristat**. Lo script si attiva da linea di comando con diverse opzioni

```
bristat [-u user] [-p] [-m] logfile1 .. logfileN
```

invocato senza opzioni **bristat** permette di avere il numero delle partite vinte da tutti gli utenti che hanno partecipato a partite registrate nei logfile **logfile1 ...logfileN**. Se è specificata l'opzione **-u user** le informazioni si limitano al solo utente **user**.

L'opzione **-p** fornisce le stesse informazioni sulle partite perse, invece che sulle vinte. L'opzione **-m** permette di avere la media (intera!) dei punti ottenuti per ogni partita vinta/persa.

Le opzioni si possono trovare in posizioni qualsiasi nella riga di comando, ad esempio le seguenti invocazioni sono equivalenti

```
bristat -u user -m logfile1 logfile2
bristat logfile1 -m logfile2 -u user
bristat logfile2 logfile1 -u user -m
```

Le informazioni statistiche vanno scritte sullo standard output secondo il formato

```
user topolino :: partite vinte: 0 media punti: 0
user nonnapapera :: partite vinte: 1 media punti: 74
Bye
```

oppure

```
user topolino :: partite vinte: 0
user nonnapapera :: partite vinte: 1
Bye
```

se non è richiesta la media. Ulteriori dettagli sul formato si possono derivare dal file **out.bristat.check** che contiene l'output del test **testbristat**.

È fatto esplicito divieto di usare **sed** o **awk** nella realizzazione dello script.

7 Makefile

È richiesto di completare il Makefile fornito nei vari kit con i target richiesti dai vari frammenti. Il Makefile consegnato con il progetto finale dovrà essere completo, e quindi contenere anche i target aggiunti nei frammenti precedenti.

8 Istruzioni

8.1 Materiale fornito dai docenti

Nei kit del progetto vengono forniti

- funzioni di test e verifica
- **Makefile** per test e consegna
- file di intestazione (.h) con definizione dei prototipi e delle strutture dati
- vari **README** di istruzioni

8.2 Cosa devono fare gli studenti

Gli studenti devono:

- leggere *attentamente* i **README** dei vari frammenti e capire il funzionamento del codice fornito dai docenti
- implementare le funzioni richieste e testarle
- verificare le funzioni con i test forniti dai docenti (attenzione: questi test vanno eseguiti su codice già corretto e funzionante altrimenti possono dare risultati fuorvianti o di difficile interpretazione)
- preparare la *documentazione*: vedi la Sez. 10.
- sottomettere i tre frammenti del progetto esclusivamente utilizzando il makefile fornito e seguendo le istruzioni nel **README**.

Gli studenti devono realizzare quanto richiesto, testarlo in modo che compili ed esegua correttamente su almeno una macchina del Centro di Calcolo Fibonacci (da specificare nella relazione) e documentarlo come specificato nelle sezioni successive.

9 Parti Opzionali

Possono essere realizzate funzionalità ed opzioni in più rispetto a quelle richieste (ad esempio altri comandi del client, black list nel server, bilanciamenti ABR, altre statistiche nello script).

Le parti opzionali devono essere spiegate nella relazione e corredate di test appropriati.

10 Codice e documentazione

In questa sezione, vengono riportati alcuni requisiti del codice sviluppato e della relativa documentazione.

10.1 Vincoli sul codice

Makefile e codice devono compilare ed eseguire CORRETTAMENTE su (un sottinsieme non vuoto del) le macchine del CLI. Il README (o la relazione) deve specificare su quali macchine è possibile far girare correttamente il codice. Inoltre, se si usano software e librerie non presenti al CLI: (1) devono essere presenti nel tar TUTTI i file necessari per l'installazione in locale del/i tool e (2) devono essere presenti nel makefile degli opportuni target per effettuare automaticamente l'installazione in locale. Se questa condizione non è verificata il progetto non viene accettato per la correzione.

La stesura del codice deve osservare i seguenti vincoli:

- la compilazione del codice deve avvenire definendo delle regole appropriate nel Makefile;
- il codice deve compilare senza errori o warning utilizzando le opzioni `-Wall -pedantic`
- NON devono essere utilizzate funzioni di temporizzazione quali le `sleep()` o le `alarm()` per risolvere problemi di race condition o deadlock fra i processi/thread. Le soluzioni implementate devono necessariamente funzionare qualsiasi sia lo scheduling dei processi/thread coinvolti
- DEVONO essere usati dei nomi significativi per le costanti nel codice (con opportune `#define` o `enum`)

10.2 Formato del codice

Il codice sorgente deve adottare una convenzione di indentazione e commenti chiara e coerente. In particolare deve contenere

- una intestazione per ogni file che contiene: il nome ed il cognome dell'autore, la matricola, il nome del programma; dichiarazione che il programma è, in ogni sua parte, opera originale dell'autore; firma dell'autore.
- un commento all'inizio di ogni funzione che specifichi l'uso della funzione (in modo sintetico), l'algoritmo utilizzato (se significativo), il significato delle variabili passate come parametri, eventuali variabili globali utilizzate, effetti collaterali sui parametri passati per puntatore etc.
- un breve commento che spieghi il significato delle strutture dati e delle variabili globali (se esistono);
- un breve commento per i punti critici o che potrebbero risultare poco chiari alla lettura
- un breve commento all'atto della dichiarazione delle variabili locali, che spieghi l'uso che si intende farne

10.3 Relazione

La documentazione del progetto consiste nei commenti al codice e in una breve relazione (massimo 10 pagine) il cui scopo è quello di descrivere la struttura complessiva del lavoro svolto. La relazione *deve rendere comprensibile il lavoro svolto ad un estraneo, senza bisogno di leggere il codice se non per chiarire dettagli implementativi. In particolare NON devono essere ripetute le specifiche contenute in questo documento.* In pratica la relazione deve contenere:

- le principali scelte di progetto (strutture dati principali, algoritmi fondamentali e loro motivazioni)
- la strutturazione del codice (logica della divisione su più file, librerie etc.)
- la struttura dei programmi sviluppati
- la struttura dei programmi di test (se ce ne sono)
- le difficoltà incontrate e le soluzioni adottate
- quanto altro si ritiene essenziale alla comprensione del lavoro svolto
- README di istruzioni su come compilare/eseguire ed utilizzare il codice

La relazione deve essere in formato PDF.